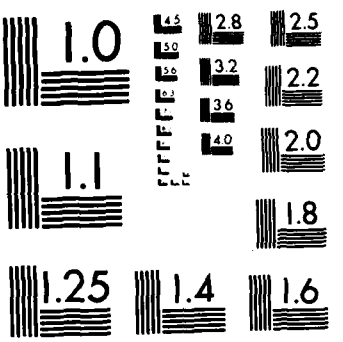
AD-A171 682 PROGRAMATICS II(U) CARNEGIE-MELLON UNIV PITTSBURGH PA 1/1
A N HABERMANN 12 MAY 81 DAAB07-82-C-J173

UNCLASSIFIED F/G 9/2 NL



MICROCOPY RESOLUTION TEST CHART

Programatics II

A. N. Habermann
Carnegie-Mellon University
Pittsburgh, Pa 15213

Contract DAAB-07-82-C-J173

Abstract

"Programatics" is the context in which one can talk about programs written in the functional programming language "Alfa". Programs written in Alfa are collections of domain and function definitions. The most important tool for defining functions is that of *functional composition*, which allows us to write functional expressions that create new functions out of existing ones. In addition to the language Alfa, Programatics contains *rewriting rules* for functional expressions and a library of proven *functional equivalences* that can be used for substitution of subexpressions. This report supersedes "Notes on Programatics Part I", dated 8 September 1980. Two major changes are that primitive types, such as integer, are no longer part of the Alfa language and that the concept of *type* has been replaced by that of *domain*.

This research is sponsored by the Software Engineering Division of CENTACS/CORADCOM, Fort Momouth, NJ.

Table of Contents

1. Introduction.

1.1 Names and Primitive Domains.

The universe of objects we are going to talk about is partitioned in a small (finite) collection of non-intersecting sets, called *primitive domains*. Examples of primitive domains are the set of all integers, the set of ASCII characters, etc. For several of the primitive domains, object names are chosen such that their representation indicates the primitive domain of the object referred to. For instance, the representation of the number twelve as a string of digits, 12, reveals that the object referred to belongs to the set of integers. Likewise, the representation of a character between single quotes, e.g., '+' or '7' or 'x', conveys that the object referred to belongs to the set of ASCII characters. In other cases, objects of a primitive domain are made public by listing their names. If objects in different primitive domains have the same name, potential ambiguities can be avoided by prefixing the object names by the names of the domains these objects belong to. For example, the name John occurs in both families

Smith := {John, Mary, Susan, Dave},
Jones := {Bob, Betty, John, Don}

Ambiguities can be avoided by using the full names "Smith.John" and "Jones.John". The primitive domains will all be referred to by distinct names.

We make one important assumption about the names we use:

> For every pair of names denoting objects in a given primitive domain, it is decidable whether or not these names denote the same object. That is, for every pair of names (u, v), denoting objects in a primitive domain D, the predicate "u = v" is computable and yields either true or false.

Examples: let T := {a, b, c, d}.

The equations " first(T) = a " and " successor(b) = c " are true.
The equations " predecessor(c) = a " and " last(T) = b " are false.

1.2 The Atomic Domain.

It is often the case that a function f(x) is defined for certain values of its argument 'x', but not for all conceivable values. Such a function is called a *partial* function. For example, the function "sqrt", which computes the square root of its argument 'x', is defined for real numbers greater or equal to zero, but not for negative numbers. The set of argument values for which a function is defined is its *input domain*. If D is the input domain of a function f(x), the set of all images, f(D), is its *output domain*.

All domains are constructed out of elements of the *atomic domain* T and the unique object *nil*. The elements of T are called the *atoms*. The atomic domain is the union of a collection of non-intersecting primitive domains, $T := B \cup C \cup I \cup \ldots$. Primitive domains that usually are included in the atomic domain T are the set B (the set of Boolean values {true, false}), the set C (the set of ASCII characters) and the set I (the set of integer numbers). Other primitive sets may be included if so desired. For example, the atomic domain may be extended by a new primitive domain "Color" or a domain "Day". New primitive domains are defined by enumerating their atoms. For example,

domain Day := (Sun, Mon, Tue, Wed, Thu, Fri, Sat)

In the next sections, the specific nature of the primitive domains is of little or no importance. We assume, for the time being, that some primitive domains have been chosen, one of which is B, the Boolean domain. All domains, including the predefined domains and the ones added by definition to the set of atoms, are *enumerable* sets. (In fact,for computer applications, primitive sets may be large, but everyone of them, including the set of real numbers, is finite.) The property of being enumerable is considered with favor, because it makes all primitive domains *totally* ordered. For every pair of objects (u, v) of a primitive domain D, the predicates "u < v", "u <= v", "u >= v" and "u > v" are computable and have the usual meaning.

The unique object *nil* is called a *sequence*, and in particular the *empty* sequence. We assume that nil is not an element of T. In the next section we will see how the empty sequence is used to create non-empty sequences.

1.3 Sequences.

A *proper* sequence (or a *non*-empty sequence) is constructed out of an existing sequence and an object that is either a sequence or an atom. The construction is denoted as "z << u", where "z" is a sequence, "u" is either a sequence or an atom, and "<<" represents the operation *push*. (Occasionally, we write "push(z, u)" instead of "z << u".) The operation *push* is defined in conjunction with the operations *pop* and *last* by the rule

IF s := push(z, u) = z << u, THEN pop(s) = z and last(s) = u.

Initially, there are no other existing sequences than the empty sequence, nil. Thus, the first collection of proper sequences that one can create, all use nil for "z" and an atom or nil for "u". If the atomic domain is defined as T := {a, b, c}, the first collection of proper sequences is

nil << a, nil << b, nil << c, nil << nil.

Since pop(z << u) = z, application of "pop" to any sequence of this collection yields nil as result. Since last(z << u) = u, application of "last" to the sequences of this collection yields respectively a, b, c and nil.

The proper sequences that we just created can now be used to generate longer sequences by substituting them either for "z" or for "u" in the expression "z << u". It is obvious that, because of the construction procedure, all proper sequences must have the form

nil << u_1 << u_2 << . . . << u_n, where n > 0.

It is frequently more convenient to represent a proper sequence as a list of the form

(u_1, u_2, . . . , u_n), for n > 0.

An alternative representation of the empty sequence, nil, consistent with the list notation is "( )".

Examples.

push	result	pop	last
nil << a	(a)	( )	a
(a) << a	(a, a)	(a)	a
(b) << a	(b, a)	(b)	b
(a, c) << (b)	(a, c, (b))	(a, c)	(b)
(a, c) << (b, c)	(a, c, (b, c))	(a, c)	(b, c)
(a, a) << nil	(a, a, nil)	(a, a)	nil

The set of all proper sequences, S, is defined as the transitive closure (T, nil, push). The union S ∪ {nil} is the set of all sequences, Z. The union T ∪ Z = T ∪ S ∪ {nil} is the *universe*, U. We adopt the convention that "t" represents an atom (an element of T), "z" a sequence that may be empty (an element of Z), "s" a proper sequence (an element of S) and "u" any object of the universe (any element of U).

1.4 Equality of Atoms and Sequences.

The fact that the set of atoms, T, is the union of a collection of non-intersecting primitive domains, implies that each atom belongs to exactly one primitive domain. In Section 1 we stated the assumption that for every pair of elements (u, v) of a set D the predicate "u = v" is computable and yields either true or false. Thus, equality is defined for every pair of elements of a primitive domain. For any pair of atoms belonging to two different primitive domains, equality is false, because primitive domains have no elements in common. For the empty sequence, nil, we define the equality "nil = nil" to be true, while the equality "nil = u" is defined to be false if u is either an atom or a proper sequence. For proper sequences s_1, s_2, equality is defined by the rule

$s_1 = s_2$ is true iff $pop(s_1) = pop(s_2)$ and $last(s_1) = last(s_2)$.

Theorem 1: Every proper sequence s can be broken apart and put together again by the rule: $s = pop(s) \ll last(s)$.

Proof. Let $z := pop(s)$ and $u := last(s)$.
The relationship between "push", "pop" and "last" states that
$pop(z \ll u) = z$ and $last(z \ll u) = u$.
The equality $s = z \ll u$ is true, because $pop(s) = z = pop(z \ll u)$ and $last(s) = u = last(z \ll u)$.
•••••

Examples. Let T := {a, b}.

a = a is true	a = b is false	a = nil is false	(a) = nil is false
(a) = (a) is true	(a (b)) = (a (b)) is true	a = (a) is false	nil = (a b) is false
(a (b)) = (a b) is false	(nil) = nil is false	nil = ( ) is true	(a a a) = (a a a) is true

Note.

The reader should be aware of the distinction between the three symbols " = ", ": = " and " = = ".

- " = " represents equality of objects in the universe U. As such, it represents a *function* (or *predicate*) that maps a pair of elements of U into the boolean domain {true, false}.
- ": = " stands for the phrase "is being defined as". The lefthand side is a name that is introduced as a representation of the righthand side.
- " = = " represents functional equivalence. Two functions "f" and "g" are equivalent on a common domain D, denoted by "f = = g", if $f(x) = g(x)$ for all $x \in D$.

1.5 Representation of Domains.

Domains are needed in function definitions for specifying the *functionality* of a function, describing the input and output domains. We use a notation of the form:

f : D -> E,

where "f" is the function name. "D" the input domain, "E" the output domain and "D -> E" the functionality of function f.

It is often the case that a domain is supposed to describe a set of objects that have a common structure, such as pairs of integers or matrices of a certain size. Primitive domains are enumerable sets and can therefore be represented as sequences. It is possible to describe other domains also as sequences, but such an approach leads to a rather clumsy notation. Take for example the case of integer pairs. The domain of all integer pairs can be constructed as a sequence by distributing the primitive domain "int" over all elements of itself, resulting in pairs (i, int), and by then distributing each first element of these pairs over the second. The concatenation of all resulting sequences is the sequence that contains all integer pairs[1]. It is obvious that this way of describing domains is not very convenient.

A better construction procedure for domains is the following. Analoguous to the construction of objects out of primitive objects and nil, domains are generated from the primitive domains by taking subsets and by combining existing domains. Starting with the primitive domains, other domains are generated by the rules:

- $(D^1, \ldots, D^n)$, for $n >= 1$, represents the domain of sequences whose i^{th} element is an object in domain D^i, for $1 <= i <= n$.
- (D . .) is a domain of sequences [including the empty sequence] whose elements are objects in domain D.
- (D, ,) is a domain of sequences [not including the empty sequence] whose elements are objects in domain D.
- D υ E is the domain of elements that are in the union of domains D and E.
- D | p is the domain of elements x in domain D for which the predicate p(x) is true.

Examples.

---

[1] Anticipating the discussion of functional composition in Chapter 2, the function that generates the sequence of all integer pairs from the primitive sequence of integers is

"link : aldistr · rdistr · (id, id) : int").

```
   nat := int | id > 0                    even := int | (id % 2) = 0
  frac := (int, int) | last # 0           arith := int ν frac ν float
   NIL := u | id = nil                     tree := NIL ν (u, tree, tree)
intvec := (int . . )                    intmatr := (intvec . . ) | = : αlen
family := (father, mother, son, daughter) parent := family | id ε (father, mother)
```

The function "id" maps the input onto itself. The predicate "id > 0" is true for positive integers and false for zero or negative integers. The function "%" is the remainder function. The domain of fractions is defined as a restriction on the domain of integer pairs by requiring that the second element be non-zero. The domain of binary trees is defined recursively. At least one of the alternatives in a recursive definition must be non-recursive. An intvec is defined to be a (potentially empty) sequence of integers. An intmatrix is a sequence of intvecs that all are of the same length. The function "αlen" takes the length of all elements of the input sequence and the function "=" compares the results for equality. The family domain is an example of a new domain (cf Section 2) and parent is defined as a subdomain by restricting the family domain.

2. Functional Composition.

2.1 Constant Functions.

A function f that maps every element of the universe onto a fixed element u ε U is called a *constant* function. For instance, the function f(x) := 5 maps every real number onto the integer number 5. Likewise, the function f(x, y) := (0, 0) maps every point in the cartesian plane onto the origin.

For every element u ε U there is exactly one constant function that maps every element of the universe onto that particular element. The constant function associated with a particular element u ε U is denoted by φ[u]. Thus, φ[u] is the function f such that φ[u](x) = f(x) = u for all x ε U.

One of the functional composition rules in Alfa is that of *serial* composition, denoted by the symbol ":". Its definition is that of functional composition in mathematics:

"f : g" is the function "h" such that h(x) = f(g(x)).

Two important properties of constant functions are expressed in the following theorems.

Theorem 1: For every function f and element u ε domain(f), the serial composition of f with the constant function φ[u] is the constant function φ[v], where v = f(u). That is,

f : φ[u] = = φ[f(u)].

Proof. Let v := f(u), g := φ[v] and h := f : φ[u].
h(x) = f(φ[u](x)) = f(u) and g(x) = φ[v](x) = v.
Since f(u) = v, h(x) = g(x) for all x ε U. Thus, h = = g.
•••••

Theorem 2: For every function f and every element u ε U, the serial composition of the constant function φ[u] with f is the same as that constant function on the domain of f. That is,

φ[u] : f = = φ[u] on domain(f).

Proof. Let g := φ[u] and h := g : f. Thus, g(x) = u for all x ε U.
h(x) = g(f(x)) = u for all x for which f is defined. Thus, h = = g on domain(f).
•••••

Examples.

composition	result function	application
sqrt : φ[16]	φ[4]	sqrt(φ[16](23)) = sqrt(16) = 4
φ[7] : sqrt	φ[7] for x >= 0	φ[7](sqrt(16)) = φ[7](4) = 7
φ[nil] : max	φ[nil] for (int, int)	φ[nil](max(3,8)) = φ[nil](8) = nil
max : φ[(2, 7)]	φ[7]	max(φ[(2, 7)](x)) = max(2, 7) = 7

END

10-86

DTIC